Tutorial: Bullseye 5

Bullseye shows you how to use the THINK Project Manager's source-level debugger. Bullseye is a simple application that draws a series of concentric circles in a small window. Its **Width** menu lets you select the width of each of the rings.

Before you begin

Make sure you're using System 7 or MultiFinder under System 6, and that you have at least 4MB of available memory.

Make sure the `Bullseye Folder` is on your hard disk, since it contains all the files you need to follow this tutorial. If you followed the directions in Chapter 2, it should be inside the `Demos` folder, which is inside your `Development` folder.

Make sure that the file `THINK Debugger` is in the same folder as the THINK Project Manager. This file must be named `THINK Debugger`.

What you should know

Before you try this example, you should know how the THINK Project Manager works. You should know how to open a project, how to edit source files, and how to run a project. If you're not familiar with any of these operations, go back and read (or try) the examples in the two chapters.

Contents

Opening the Bullseye Project 71

Turning the Debugger On 71
- Generating the debugging tables 72
- Running the project 72

Watching the Program Run 72
- The Source window 73
- Stepping through statements 74

Stepping into functions 75
Stepping out of functions 75
Tracing every statement 76
Setting a breakpoint 77
Letting the program run 79
Stopping the program 79
Viewing other files 79

Examining and Setting Variables 80
The data window 81
Examining variables 81
Changing the value of a variable 85

Examining structs, classes, and arrays 86

Expressions and Contexts 90
How and when the source debugger evaluates expressions . 91
Display formats 91

Quitting the Debugger 93

Opening the Bullseye Project

If you're at the Finder, double-click the `bullseye π` project in the `Bullseye Folder`. If you're already in the THINK Project Manager, use the **Open Project** command in the **Project** menu to open the `bullseye π` project.

Bullseye consists of three source files and the MacTraps library, as shown in Figure 5-1.

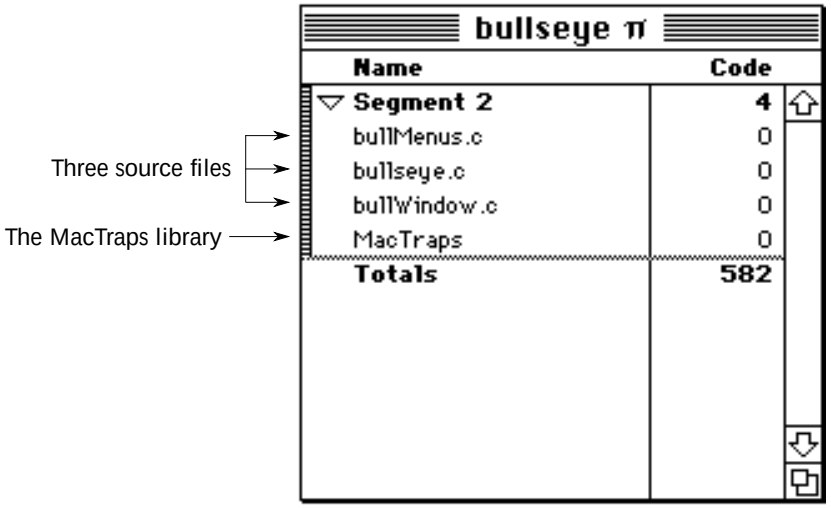

Figure 5-1 The bullseye π project window

Note that none of the files has been compiled and that the MacTraps library hasn't been loaded.

Turning the Debugger On

The THINK Project Manager ordinarily runs your project without the debugger. Choose the **Use Debugger** command from the **Project** menu.

When the source debugger is on, the THINK Project Manager adds a "bug" column in the project window to the left of the Name column.

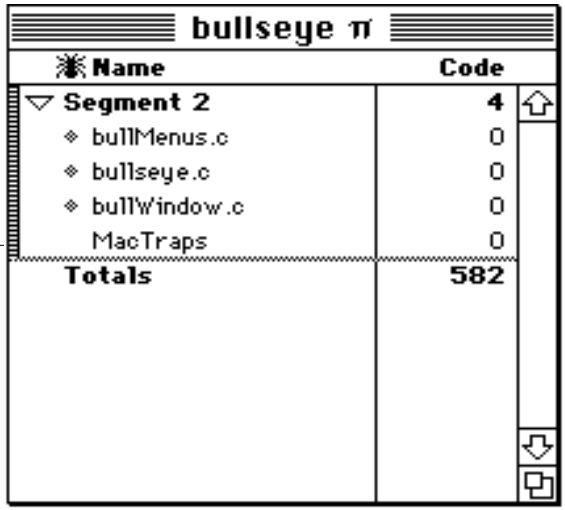

Figure 5-2 The bullseye π project window with the debugger on

Generating the debugging tables

The gray diamonds in the bug column indicate that the THINK Project Manager will generate special debugging tables for a source file. These tables go into your project document along with the source files' object code. The THINK Project Manager never generates additional code when you run the debugger.

Running the project

Choose **Run** from the **Project** menu to let the THINK Project Manager compile and load all the files in your project. The THINK Project Manager will generate debugging tables for all the files as well.

Watching the Program Run

Instead of running your project, the THINK Project Manager launches the source debugger, which controls the execution of your program. The debugger displays two windows at the bottom of your screen. If you're using two screens, the debugger windows are on the second screen instead.

The window on the left is the Source window. It contains the source text of your program. The window on the right is the Data window.

You use this window to examine and set the values of your variables as you debug your program.

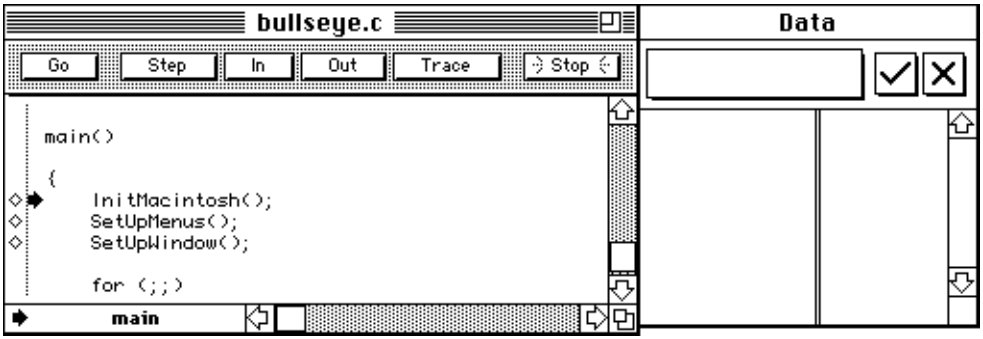

Figure 5-3 THINK debugger

The Source window

The Source window shows you the source of your program. The title of the window is the name of the source file you're looking at.

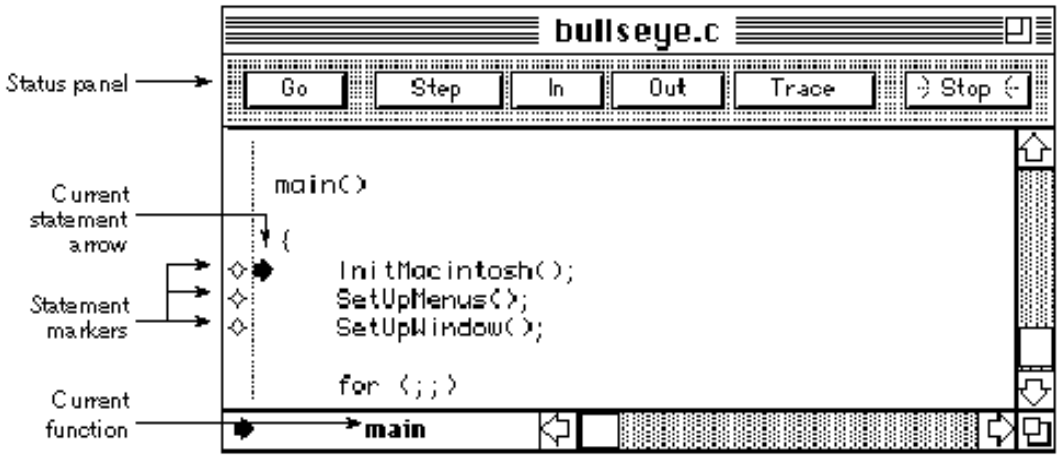

Figure 5-4 Source window of the THINK debugger

The Source window shows the source text of your program. When you start the debugger, this window shows the file that contains the `main()` routine of your application.

The top of the Source window has a six-button status panel. These buttons control the execution of your program. The names of these buttons match the commands in the debugger's **Debug** menu. In the

figure, the Stop button is lit, which indicates that your program is stopped.

The black arrow to the left of the first line of the program is the current statement arrow. This indicator shows you the current statement, the one that the debugger is about to execute.

The column of diamonds running along the left side of the Source text are statement markers. Every line of your program that generates code gets a statement marker. Later, you'll use the statement markers to set breakpoints.

The source debugger uses the space at the lower left of the source window for the name of the current function. When you click here and hold the mouse button down, the debugger displays a pop-up menu that shows the call chain—the names of the functions that were called to get to the current function.

Stepping through statements

Click the Step button in the status panel, as shown in Figure 5-5.

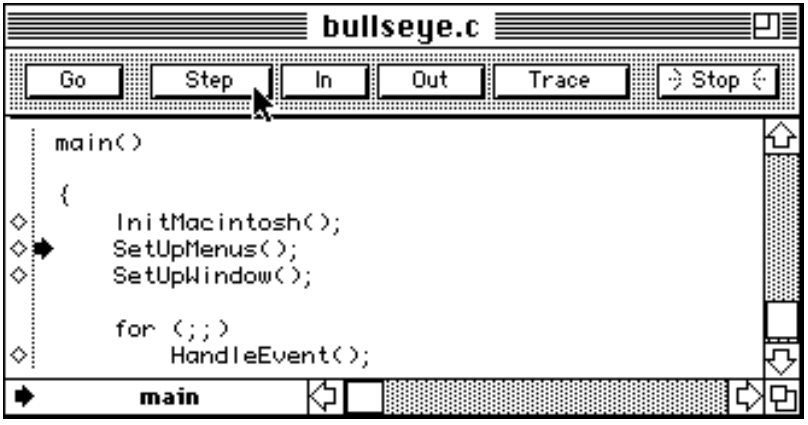

Figure 5-5 Stepping through the program

The Step button lights up for a moment, the current statement arrow moves to the second statement, and the program stops again.

The Step button lets you execute your program line by line. You can use the **Step** command in the **Debug** menu or press Command-S to do the same thing.

Click the Step button (or press Command-S) one more time so the current statement arrow points to the call to `SetUpWindow()`. This function creates the window that Bullseye uses.

Stepping into functions

*Instead of clicking the In button, you can also choose the **Step In** command from the **Debug** menu or press Command-I.*

To see how `SetUpWindow()` works, click the In button on the status panel.

Now the current statement arrow points to the first line of the `SetUpWindow()` function. This function is in the file `bullWindow.c`, so the title of the window changes to let you know what file it's displaying.

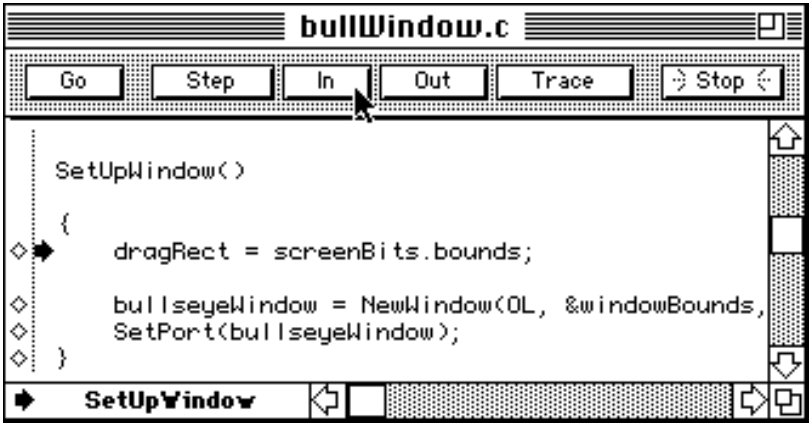

Figure 5-6 Inside SetUpWindow()

Note

The current statement arrow doesn't have to be right before a function call for the In button to work. The **Step In** command executes every statement until the program counter is no longer in the current function. Another way to think of the **Step In** command is: "Keep going until you fall into a function." (**Step In** also stops execution if you fall out of the current function.)

Stepping out of functions

You can see the entire `SetUpWindow()` function in the Source window. This is a straightforward function.

Click the Out button to leave the `SetUpWindow()` function.

The Source window now shows that the debugger is ready to execute the function `HandleEvent()` in the `bullseye.c` file.

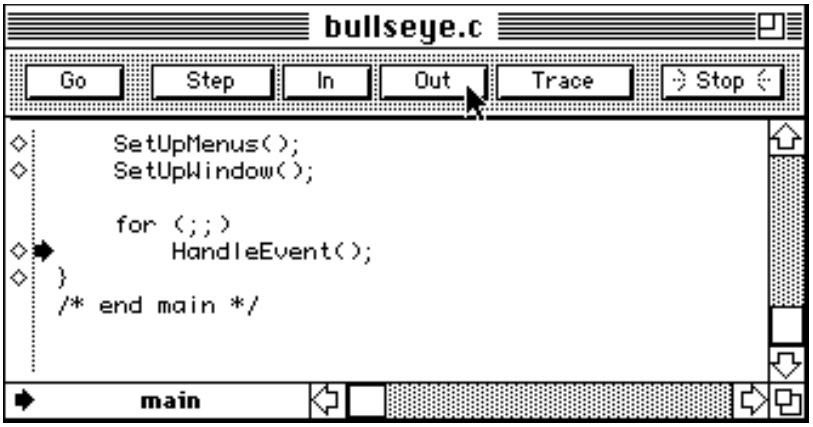

Figure 5-7 Outside SetUpWindow()

The Out button steps through each statement in the current function until the current statement arrow leaves the function.

Tracing every statement

*Instead of clicking the Trace button, you can choose the **Trace** command from the **Debug** menu or press Command-T.*

Now click the Trace button.

The current statement arrow now points to the first statement of the `HandleEvent()` function.

Figure 5-8 Inside HandleEvent()

Tracing takes you to the next statement even if it has to step into a function. If you were to continue tracing, you'd stop at every statement. Stepping, on the other hand, *never* dives into a function.

Note

The In button actually does a trace until the current statement arrow leaves the current function.

Setting a breakpoint

Since the `SetUpWindow()` function opened a new window, the program will get an activate event the first time through the event loop. In Bullseye, all the program does on activate events is call `InvalRect()` on the whole window, so the second time through the event loop it gets an update event.

You could **Step** or **Trace** to verify that this is what really happens. A faster way is to set a breakpoint at the function that redraws the window.

Scroll down in the Source window until you get to the code that handles update events. Click the statement marker to the left of the `DrawBullseye()` function.

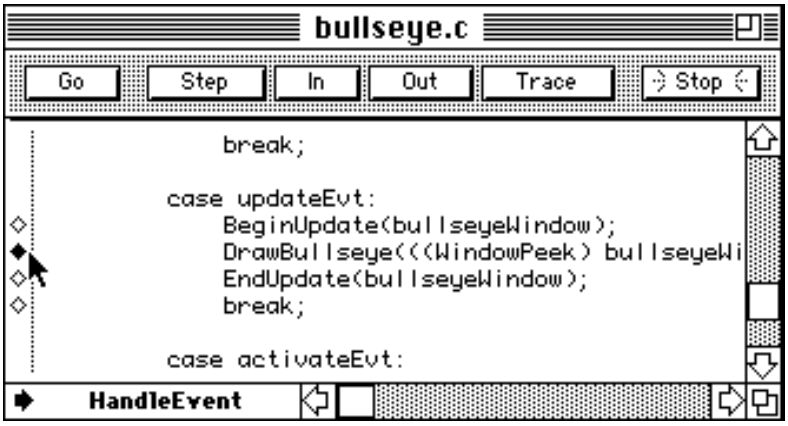

Figure 5-9 Setting a breakpoint

The hollow diamond turns black to indicate that you've set a breakpoint. You can set as many breakpoints as you like this way. When your program is about to execute a statement that has a breakpoint, it will stop. To remove a breakpoint, click the filled diamond.

To start your program running, click the Go button. The program runs for a few moments and then stops. The current statement arrow is at your breakpoint.

Click the In button to step into the `DrawBullseye()` function. (This function is in the `bullWindow.c` file, so that's the file you see in the Source window now.)

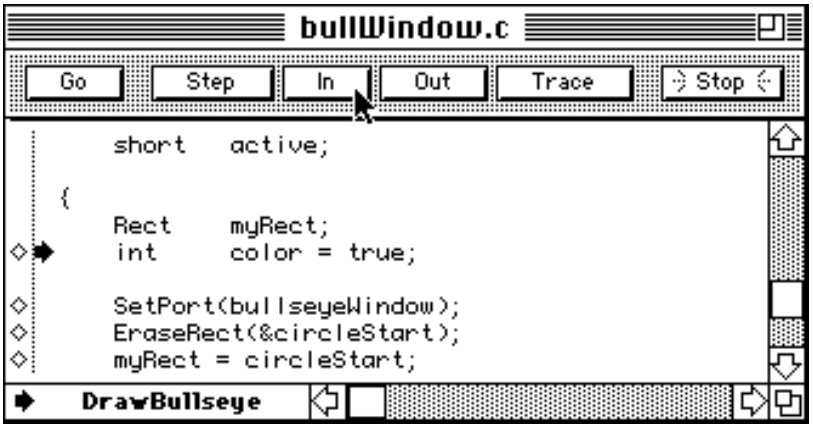

Figure 5-10 Inside DrawBullseye()

Click the Step button to watch how the program draws a bullseye in the window. To stop, click the Out button. Whether you **Step** or **Step Out**, you eventually end back at the call to `DrawBullseye()`.

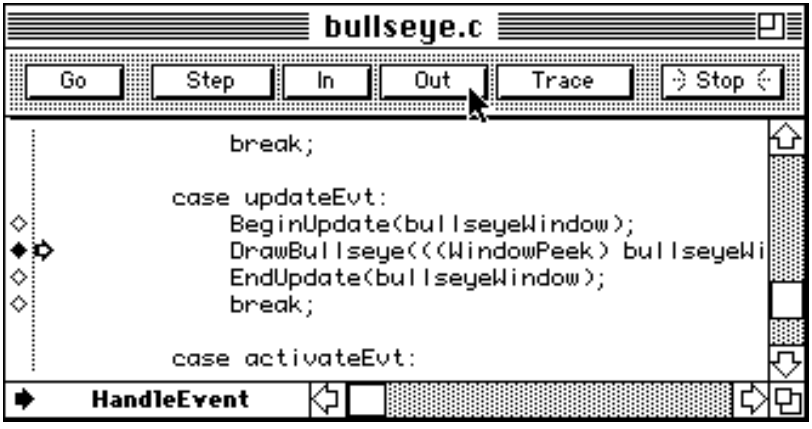

Figure 5-11 Outside DrawBullseye()

Note that the current statement arrow is hollow. This means that there are still some instructions left to execute in the statement. You see hollow arrows when the statement is making an assignment or cleaning up the stack after stepping out of a function.

Before you go on, clear the breakpoint. Just click the diamond.

Letting the program run

Click the Go button to let the program run. You can set and clear breakpoints while your program is running.

When you click in the Source window to set breakpoints, your application goes to the background and the debugger comes to the foreground. If you click the Go button when your program is running, the debugger brings it to the foreground.

Stopping the program

To stop your program, click the Stop button or, if the debugger is the frontmost application, press Command-Period. Your program will stop as it's coming out of one of the event-fetching routines (`GetNextEvent()` or `WaitNextEvent()`).

If your program is stuck in a loop, or if you want to stop it without waiting for control to return to the event loop, you can use the panic button—Command-Shift-Period—to stop your program. Be careful when you do this, though, because the debugger will stop execution no matter what it's doing.

Viewing other files

The Source window usually shows the file that contains the current statement. To look at another file in the Source window (to set breakpoints in it, for example), you tell the THINK Project Manager to send the text to the debugger:

1. Click the project window.
2. Click the name of a file.
3. Choose **Debug** from the THINK Project Manager's **Source** menu.

4. The file that you chose appears in the Source window.

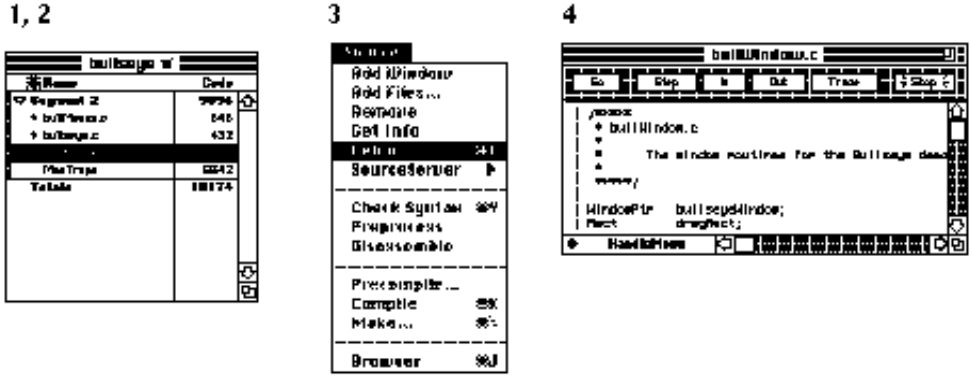

Figure 5-12 Viewing another file in the Source window

Now you can examine the file and set breakpoints in it. To get back to the current file, click the current function indicator at the lower left of the Source window or press the Enter key.

Examining and Setting Variables

Tracing your program's execution lets you see what your program is doing. But to really fix bugs, you need to be able to examine your variables. You use the Data window for this task.

If you've quit the Bullseye program, start it up again. Choose **Run** from the **Project** menu, and when you see the debugger window, click the Go button in the status panel.

The data window

The Data window appears to the right of the Source window. The best way to think about this window is to treat it as a spreadsheet.

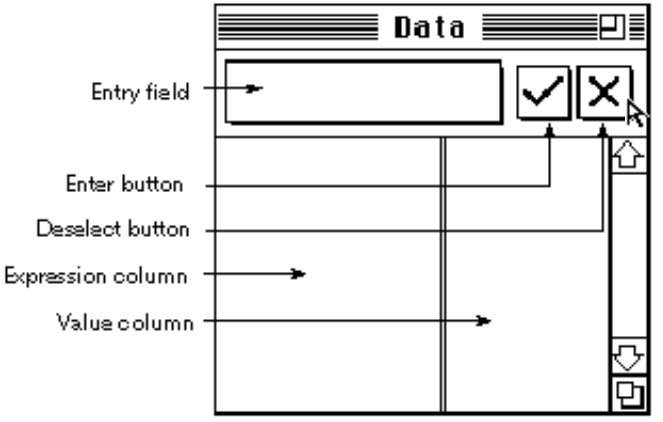

Figure 5-13 Data window of the THINK debugger

Expressions you type into the entry field appear in the left column when you click the Enter button (check mark) or when you press the Return or Enter key. Pressing the Enter key leaves the expression selected. Pressing the Return key leaves the entry field empty so you can type the next expression. The Enter button works just like the Enter key.

If you change your mind and don't want to enter an expression, click the Deselect button (X mark). The expressions you enter appear in the left column and their values are displayed in the right column.

You can drag the center bar to make a column wider.

To clear an expression in the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key.

Examining variables

Suppose you want to watch the value of the `menuID` variable in the `HandleMenu()` function. First, make sure the `bullMenus.c` file is displayed in the Source window. If it's not, bring the Project window to the front, click the name `bullMenus.c`, and choose **Debug** from the **Source** menu.

Next, scroll down until you see the `HandleMenu()` function, and set a breakpoint at the `switch` statement. Remember that you can set breakpoints even while your program is running.

After you set the breakpoint, click once on the line that contains the `switch` statement to select it.

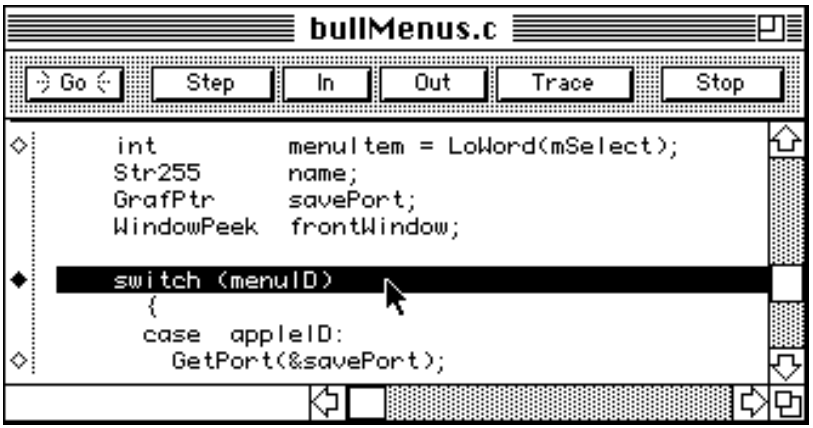

Figure 5-14 Selecting the context for the debugger

You select a line to give the debugger a context for evaluating `menuID`. In this case, you're saying that you want to know the value of `menuID` right before the `switch` statement.

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage—in other words, if it refers to nonstatic variables local to a function. All other expressions have global scope.

Click the Data window. You'll see the insertion point blinking in the entry field. Type `menuID` in the entry field and press the Return key.

The debugger compiles the expression (it takes about a second) in the context of the selected line. Right now, the Data window doesn't show a value for `menuID` because the program isn't stopped there.

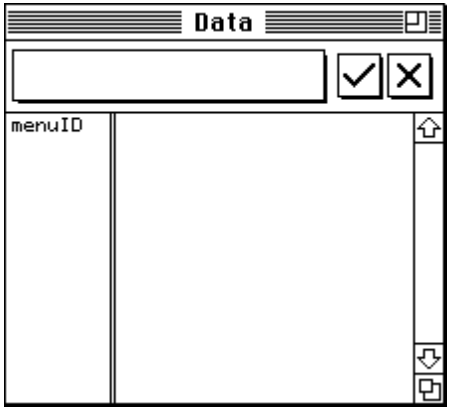

Figure 5-15 Entering menuID into the Data window

Now, go back to the Bullseye program. Click the Bullseye window, and click the Go button (or press Command-G) to bring Bullseye to the foreground.

Next, choose **7** from the **Width** menu. Your program stops at the breakpoint when you release the mouse button, and the value of `menuID` appears in the value column.

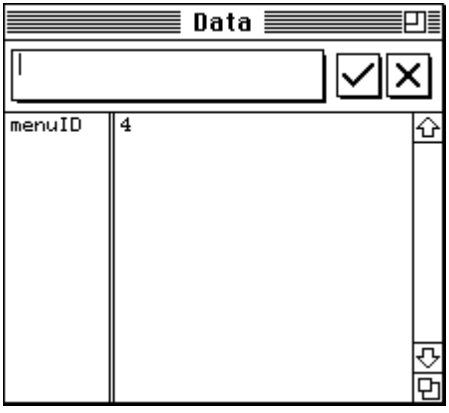

Figure 5-16 Examining the value of menuID

Any time your program stops, the source debugger displays the values of expressions that have global scope. Then it displays the values of expressions with local scope whose context is the same as the current function. Finally, the debugger clears the values of local expressions whose context is not the current function.

Changing the value of a variable

Click in the Data window again and type `menuItem`. This variable contains the item number of the selected menu item. When you press the Return key, the source debugger shows you its value.

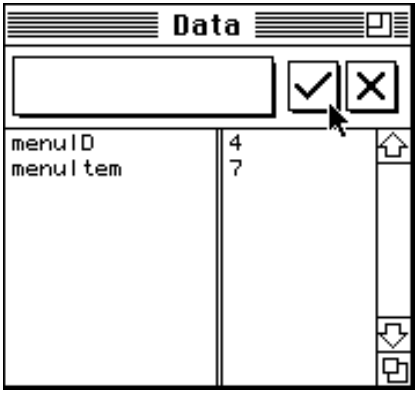

Figure 5-17 Entering menuItem in the Data window

To change the value of a variable, click its value and type a new one in the entry field. When you click the Enter button, the value of the variable changes. Here's an example:

Click the value of `menuItem` (the right column) to select it. Its value appears in the entry field as well. Now type `8` as a new value for it. Click the Enter button to assign the new value to the variable.

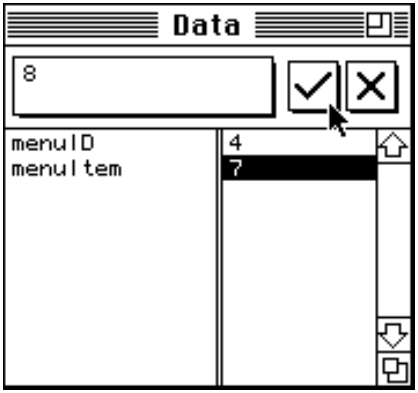

Figure 5-18 Changing the value of menuItem

When you click the Go button, the Bullseye program behaves as if you had chosen **8** from the **Width** menu.

To remove an expression from the Data window, select it and choose **Clear** from the **Edit** menu or press the Clear key.

Note

You can enter the same expression more than once in the Data window. You might want to do this to lock one of the expressions so you can compare it to the same expression later in the program. See the section “How and when the source debugger evaluates expressions” later in this chapter.

Now choose the **Clear All Breakpoints** command from the **Source** menu to make sure there aren’t any breakpoints set before you go on to the next section. Then click the Go button to start the program running again.

Examining structs, classes, and arrays

The Data window lets you examine and modify structures, classes, and arrays, not just simple variables. When you display a structure or union in the Data window, its value appears as `struct 0x000000` or `union 0x000000`. Arrays appear as `[] 0x000000`. (The real address appears instead of `0x000000`, of course.)

When you double-click one of these values, the debugger displays another window for the structure or array.

Note

Any information presented here about structures applies to unions and classes as well.

To see how this works, make sure the Bullseye program is still running. Display the file `bullseye.c` in the Source window, and

set a breakpoint on the line right after the call to `GetNextEvent()` in the function `HandleEvent()`.

Figure 5-19 Setting a breakpoint in HandleEvent()

Now click the Bullseye window. The program stops at the breakpoint. When it does, type `theEvent` in the entry field of the Data window and press the Return key.

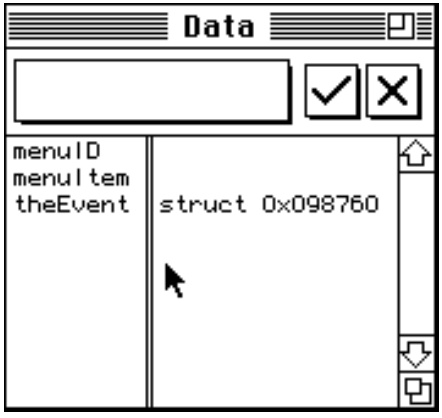

Figure 5-20 Entering theEvent in the Data window

The debugger displays the word `struct` and the address of the structure. If you can't see the entire value, click the center separator bar and drag it to the left. An an alternative, you can make the window larger.

Note

If you don't select a line to give a variable a context, the debugger uses the current statement.

Double-click the value of `theEvent`. The debugger displays a window. The names on the left are the fields of the struct.

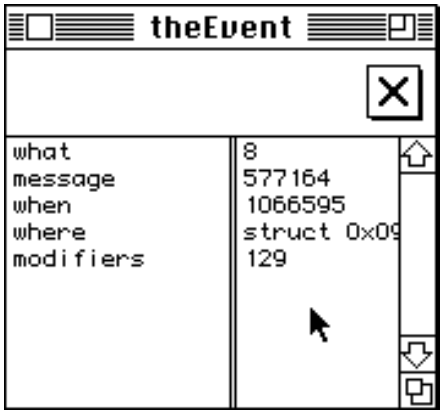

Figure 5-21 Entering theEvent in the Data window

You can edit the values of the fields, but you can't edit the names.

The `what` field indicates that you're looking at an activate event (`activateEvt = 8`). In activate events, the `message` field points to the window record that gets the activate event.

Double-click the `message` field. The debugger enters a new expression in the main Data window: `theEvent.message`.

Note

Double-clicking the left column of any Data window creates a new entry in the main Data window.

Edit the expression to read `*(WindowPtr)theEvent.message` so you can look at the `WindowPtr`.

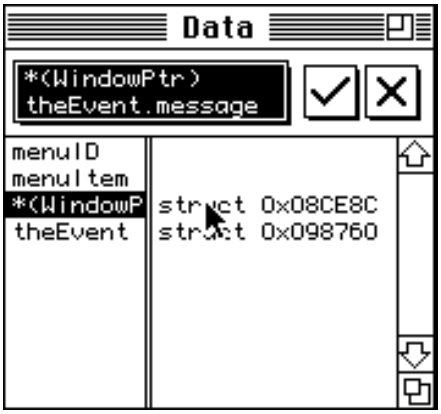

Figure 5-22 Entering ∗(WindowPtr)theEvent.message

Double-click the value of the new expression. The debugger displays another Struct window.

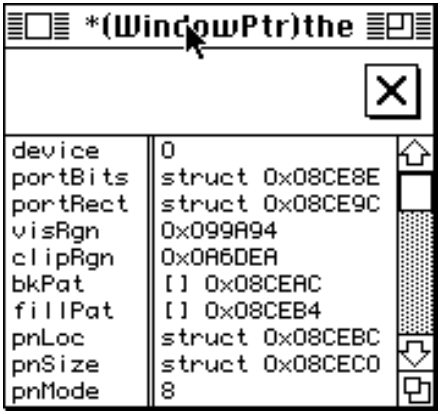

Figure 5-23 Examining the fields of ∗(WindowPtr)theEvent.message

Scroll down to the pnPat field and double-click. You'll see an Array window.

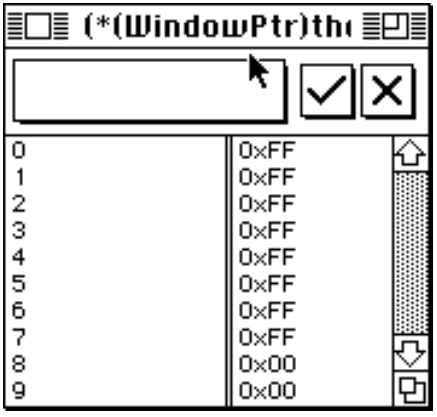

Figure 5-24 Array *(WindowPtr)theEvent.message.pnPat

Because C and C++ compilers don't enforce array bounds, Array windows have "infinite" scroll bars. Unlike structures, you can select an index in the left column and change it. When you do so, the window shows the array from the index you entered.

When you double-click the value of a pointer variable, the debugger inserts a dereferenced expression in the Data window and displays its value. To see a pointer as an array, change its format to Address.

To get rid of a Struct or Array window, you can click its close box, press the Clear key, or choose **Clear** from the **Edit** menu. When you use the Clear key or the **Clear** command, the debugger removes the expression associated with the window from the main Data window. If you click the window's close box, only the window goes away; the expression in the main Data window remains.

Expressions and Contexts

You can enter any expression that does not have a potential side effect. That means you cannot enter assignment statements or expressions involving ++, --, or +=.

Every expression you type in the entry field is compiled in a context. The context is the selected line of the Source window. If no line is selected, the context is the line to which the current statement arrow points.

To see the context of an expression, click the expression in the left column of the Data window and choose **Show Context** from the **Data** menu. The source debugger will display the context in the Source window.

To change the context of an expression, click the Source window at the line you want to use as a context. Then select an expression in the Data window and choose **Set Context** from the **Data** menu. A shortcut is to hold down the Option key as you click the Enter button.

If you edit an expression, its context will be that of the original expression. You can change its context by holding down the Option key as you click the Enter button, as described above.

How and when the source debugger evaluates expressions

Expressions in the Data window have either local scope or global scope. An expression has local scope if it refers to variables with dynamic storage—in other words, if it refers to nonstatic variables local to a function. All other expressions have global scope.

Every time your program stops, the debugger evaluates the expressions in the Data window. It displays the values for expressions with both global and local scope. The values of expressions that don't have a global or local context are cleared to make the window less cluttered.

If you want to make sure that the debugger doesn't redisplay a value, select it and choose **Locked** from the **Data** menu. A small lock icon appears next to the expression. This command is useful if you want to compare the value of the same expression at different times. You can also lock expressions to keep their values from being cleared when they go out of scope.

Display formats

The way the debugger displays expressions depends on their type. You can change the format with the formatting commands in the **Data** menu. To change a format, select an expression from the Data window. Then choose the format from the debugger's **Data** menu.

Not all formats are available for all types, as shown in Table 5-1. Defaults are in italics.

Type	Formats Available
integers	*Signed Decimal*, Unsigned Decimal, Hexadecimal, Character
unsigned	*Unsigned Decimal*, Signed Decimal, Hexadecimal, Character
pointers	*Pointer*, Address, Hexadecimal, C String, Pascal String
arrays	*Address*, C String, Pascal String
structures	*Address*
unions	*Address*
classes	*Address*
functions	*Address*
floating point	*Floating Point*

Table 5-1 Display formats available

The display formats look like this:

Format	Example
signed decimal	`4523345, -1`
unsigned decimal	`4523345, 65535`
hex	`0xA09E1487`
char	`'c', 'TEXT'`
C string	`"abcdef\nghi\33"`
Pascal string	`"\pabcdef\nghi\33"`
pointer	`0x7A7000` or `0x007A7000`
address	`[] 0x09FE44, struct 0x08FC14`
floating point	`1961.0102`

Table 5-2 Display format examples

The C string and Pascal string formats display nonprinting characters in backslash form. Whenever it can, the debugger uses the built-in

escape characters (`\n`, `\r`, `\a`); otherwise, it uses `\nn`, where `nn` is an octal value.

Of course, you can use typecasting to use formats that aren't normally available. For example, if you really wanted to see the integer `i` as a C string, you would type this expression: `(char *)i`.

To see any pointer as an array, just change its format to Address. Then, when you double-click its value, you see an Array window instead of the value of what the pointer points to.

Quitting the Debugger

The best way to quit the debugger is to quit your application. You should use the **ExitToShell** command in the debugger's **Debug** menu only when you can't use your application's **Quit** command.

5 Tutorial: Bullseye